BigDAWG: a Polystore for Diverse Interactive Applications

Adam Dziedzic, Jennie Duggan, Aaron J. Elmore, Vijay Gadepally, and Michael Stonebraker

Abstract—Interactive analytics requires low latency queries in the presence of diverse, complex, and constantly evolving workloads. To address these challenges, we introduce a polystore, BigDAWG, that tightly couples diverse database systems, data models, and query languages through use of semantically grouped "Islands of Information". BigDAWG, which stands for the Big Data Working Group, seeks to provide location transparency by matching the right system for each workload using black-box model of query and system performance. In this paper we introduce BigDAWG as a solution to diverse web-based interactive applications and motivate our key challenges in building BigDAWG. BigDAWG continues to evolve and, where applicable, we have noted the current status of its implementation.

Index Terms—polystore, federated database system, distributed query processing, data migration

1 Introduction

Over the past decade, the database community has realized the vision of "one size does not fit all" by offering specialized database engines that excel at specific application requirements [15]. Examples of such targeted systems include column stores for analytics, main-memory databases for high-throughput transactional workloads, array databases for numerical computation, and geospatial databases for location-based services. For interactive applications, selecting the right database engine is critical for ensuring that query response times are sufficiently fast. At the same time, there has been a surge in the number of database systems available and selecting the right one for a given workload is a non-trivial task. Also, a panacea database is unlikely to exist for these heterogeneous workloads because many will include multiple data models, such as combining results from text analysis with data from a streaming system.

Interactive analytics (IA) further complicate this database selection problem because their queries are ad-hoc and unlikely to conform to a static pattern amenable to manual tuning by a database administrator. If and when an IA workload evolves, the database best suited for it may change. This triggers data migration, a manual and difficult process that often requires an *extract, transform, and load (ETL)* pipeline. Such processes can be brittle as data schemas evolve and each pair of systems will need a distinct ETL process. Usually this step migrates data between systems in batches at fixed intervals, and such processes are not responsive to user needs. Given that a workload evolution may be short-lived, setting up a long-term ETL process to migrate data between systems may waste both developer and system resources. Ideally, such a data transformation between systems would be adaptive, automated, and fast.

In light of these diverse database offerings paired with limited application developer expertise on database engine selection, we submit that the time has come for a *polystore* system that couples disjoint database engines by providing location transparency to users and a unified API to support it. While similar to federated database systems [14, 2, 4] that enable query processing over a set of relational databases, a polystore, like BigDAWG [7], differs in several ways. First, many federated systems targeted a single data model, often the relational model. In BigDAWG, we explicitly consider multiple data models. Second, prior federated system research sought to incorporate several engines under different administrative domains. With BigDAWG, we assume control over each of the underlying engines and how data is mapped to each system. Recent polystore projects integrated analytic relational systems with Hadoop [12, 6], but they addressed long-running batch analytics and not IA.

As BigDAWG supports many different data models and query languages, we will explore how to provide access to the underlying engines without requiring the user to learn the interfaces for each engine, or for them to explicitly specify how data objects are mapped to specific engines. If we were to provide a single interface and data model across all engines, while supporting location transparency, the result would be an uninteresting intersection of the engines that resembles a key-value store. This arises from the need of each engine to support a common data model and API, if the user is to abstract away where the data resides. If not, semantic ambiguities between engines, such as how a system handles *NULL* values or `like` queries, would create inconsistent results.

To address this tension between providing data transparency and the need for a unified API, BigDAWG introduces the concept of an *Island of Information*. Each island describes a data model, query language or operators, and set of underlying database engines that support these semantics. Example islands include ones for relational, text, arrays, or spatial semantics. With islands a user will declare their expected semantics for a query – without having to worry about the specifics of how the data is loaded into the island or the details of interacting with a specific engine. In Section 2 we discuss how islands are composed and how users interact with islands.

An important use case for BigDAWG is to support interactive applications, and in particular web-based ones. To highlight the effectiveness of BigDAWG as an interactive system, we have constructed a demonstration [8] built upon six different interactive modules for users to explore and analyze a complex medical dataset collected from an intensive care unit [13]. To support a wide range of front-end use cases and back-end storage engines, we utilize a JSON based REST API for users to query and interact with the system. Section 2.3 outlines this API. To support low-latency queries required by interactive applications, BigDAWG must continually monitor how various queries perform on the underlying engines, and migrate data to the appropriate engine with minimal disruption to the user. As we have full control of the data placement and are not limited to one copy of a data object, we will also explore self-managed replication of objects between engines. Here, BigDAWG will need to consider the trade-offs of increasing data loading costs, and the ability to shed load between engines and proactively exploring the performance characteristics of each engine. Section 3 discusses these and other design goals for building BigDAWG.

2 BigDAWG Architecture

BigDAWG supports complex queries for IA using a new architecture for data storage and retrieval designed around data independence. It

- *Adam Dziedzic and Aaron J. Elmore are with the University of Chicago. E-mail: {adam.dziedzic,aelmore}@cs.uchicago.edu.*
- *Jennie Duggan is with Northwestern University. Email: jennie.duggan@northwestern.edu.*
- *Vijay Gadepally is with Lincoln Laboratory and Michael Stonebraker is with CSAIL at MIT. Email: vijayg@mit.edu, stonebraker@csail.mit.edu.*

Manuscript received 31 Mar. 2015; accepted 1 Aug. 2015; date of publication xx Aug. 2015; date of current version 25 Oct. 2015. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

consists of four distinct layers as shown in Figure 1: Database and storage engines; Islands; BigDAWG API; and Applications.

These layers provide visualization tools and developers with a uniform interface to a variety of database and storage engines. For example, in the notional architecture described in Figure 1, a visualization would only need to communicate with the the BigDAWG API to access a potentially large number of islands and database engines.

2.1 Database and Storage Engines

Complex interactive analytics will draw from a variety of database and storage engines. Database and storage engines make up the base layer of the architecture shown in Figure 1. To support complex queries and data models, BigDAWG does not enforce any requirements on the underlying database engine. To integrate a database with BigDAWG, one or more *shims* are needed to integrate and potentially translate data from the database to BigDAWG.

To support interactive queries where low latency and high performance are key, BigDAWG also provides support for newer streaming databases, such as S-Store [3]. In the current implementation of BigDAWG, we also support a diverse set of database engines: SciDB [1], PostgreSQL [16], Accumulo [9], and MyriaX [11].

2.2 Islands of Information

The BigDAWG API connects to different database and storage engines via a logical layer known as an *island*. An island of information is an abstraction for the functionality, programming and query language, and data model of underlying database engines. Each island is composed of a set of databases, and a shim to communicate with each database. If a database belongs to more than one island, then a shim per island is required to ensure that this database adheres to each island's semantics. These shims connect databases to the API layer and take care of moving information and queries across islands and engines. The use of semantic islands and shims is similar to the use of mediators and wrappers from earlier federated systems [2, 4, 17]. However, these earlier techniques focused on domain-specific functionality, or views, and not on data models that may be disjoint or partially overlapping.

A cross-database island supports the intersection of functionality between underlying database engines. BigDAWG on the other hand, provides support for the union of operations supported by all islands. For example in Figure 1, the array island is a cross-system island that provides an array data model and query interface to applications using BigDAWG to connect to relational database (RDBMS) X or to array database (DBMS). However, RDBMS X is also accessible through the relational island or array island in order to expose another data model or programming interface – such as a spatial island if RDBMS X supports spatial indexes and objects. For database engines without existing island support, BigDAWG supports the creation of *degenerate islands* that provide support for a single engine.

BigDAWG currently supports four cross-system islands and a number of degenerate islands. In addition to array and relation islands, the Dynamic Distributed Dimensional Model (D4M) [10] and Myria [11] islands provide two different interfaces to interacting with an overlapping set of database engines. For example, D4M supports an associative array data model and queries that are written using linear algebraic operations. Myria, on the other hand, supports a relational data model augmented by iteration and queries that are written with this extension of relational algebra. Both D4M and Myria have shims that connect to their supported database engines.

2.3 BigDAWG API Layer

The BigDAWG API layer consists of server- and client-facing components. BigDAWG incorporates many different islands which connect to database engines via shims. Clients interact with these islands through two important operations: SCOPE and CAST.

2.3.1 User-Facing Operations

Islands provide a particular language and interface to database(s). However, not all interactive analytics can be completed within a single island. Instead, BigDAWG allows users to express their query using any combination of islands. In order to provide a uniform view of the data and query, BigDAWG will move data or intermediate results between engines and islands as necessary. To specify a particular island of information, a user indicates a SCOPE in their query. A SCOPE allows users to control the data model and programming interface with which they wish their queries to be executed. A cross-island query may have multiple scopes to indicate the desired semantics of its parts. For queries that rely on cross-island interaction, BigDAWG also offers CAST operations that can automatically move data between database and storage engines – and subsequently between islands. For example, a user may wish to correlate information stored in an array, *A*, with data in a relational database, *R*, using a relational island. To execute such a query, the user can issue the full query with the RELATIONAL SCOPE and CAST data from *A* to the relational database. For example:

```
  RELATIONAL(SELECT * FROM R, CAST(A,
relation) WHERE R.v = 5 and R.id = A.id);
```

The above query produces a result that, from a user's perspective, looks like the entire array *A* was moved into a relational table in order to execute. Further, by specifying that the context (via SCOPE) of the query as RELATIONAL, the user has explicit control over the type of operation and result obtained. For example, issuing a JOIN with a RELATIONAL scope may yield different results than issuing a JOIN with an ARRAY scope.

In the current implementation, BigDAWG supports numerous islands. Of these islands, four are cross-system islands and the remaining are degenerate islands for each supported database engine.

2.3.2 Detailed API

One of the large challenges in the BigDAWG API is supporting queries that are agnostic to the underlying database engine. The API makes use of a REST Java service for common database interactions; we support *Query* and *Alert Registration* for streaming services. In order to integrate easily with applications and visualizations we use JSON-formatted data.

The Query command is a JSON object that takes a number of required and optional inputs. Table 1 shows the query and response API a client would use. The `query` parameter is a string that contains one or more SCOPE operators, and zero or more CAST operators.

The Alert Registration can be used for streaming databases such as S-Store to register instances where alerts should be passed through BigDAWG. We envision such a service being used for visualization and IA that look at streaming data and alert users when certain conditions are met (or not met). The current version of the API is also described in Table 1.

To support interactive visualization, the Query API allows for application writers to list the number of `Tuples per Page` returned by a query with optional additional pagination parameters. Although we will support an `interactive` flag to indicate the query should return quickly, we are working on expanding user-provided hints for estimating the approximate size and/or latency of results. For example, certain queries return a very large number of entries which may be undesirable for IA.

We are also looking at ways to improve performance by caching

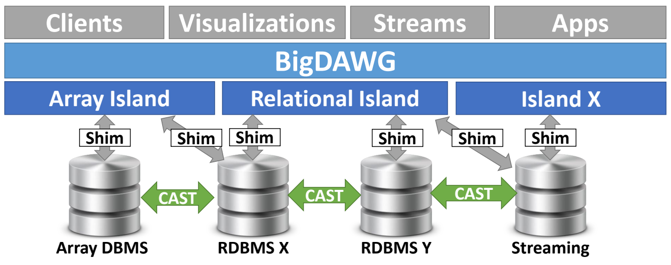

Fig. 1. The BigDAWG Architecture.

Name	Type	Req	Description
Query API			
Query	String	Yes	Query with CAST and SCOPE operations
Authorization	Object	Yes	User Authorization
TuplesPerPage	Integer	No	Number of tuples per page
PageNumber	Integer	No	Request for particular page
ResultsAsOf	Timestamp	No	Results as of a particular time
Interactive	Boolean	No	Hint for quick results
Query Response			
Response Code	Integer		HTTP response code
Tuples	List		A JSON list of tuples
Number of Tuples	Integer		Total number of tuples
Page Number	Integer		This page number
Number of Pages	Integer		Total number of pages
Schema	List		A list of <name, type >values
Timestamp	Timestamp		The freshness of the results
Alert Registration API			
Query	String	Yes	Query that system can provide alert on
NotifyURL	String	Yes	URL to call when alert is made
Authorization	Object	Yes	User Authorization
OneTime	Bool	Yes	Is this a one time alert?
Alert Registration Response			
Response Code	Integer		HTTP response code
Status URL	String		URL to check for results

Table 1. The BigDAWG client query and alert registration API.

certain results at the BigDAWG middleware level and using the polystore for load balancing across engines. The current implementation of CAST is done via use of CSV export and import into temporary tables. We are currently investigating ways to cast data directly from one engine to another via a more efficient binary loading process. The following section expands on our future design goals of BigDAWG.

3 Challenges and Design Goals

In building BigDAWG, we have identified several key challenges, including island discovery and query decomposition across multiple engines [7]. While these goals are important for a wide class of usage patterns, for a polystore to be effective for IAs it is especially critical to identify the ideal database for the interactive workload, migrate data to the appropriate engine when workload or requirements change, and minimize the overhead of having data flow through the middleware translators. In this section we outline our research goals for enabling effective IAs in BigDAWG.

For our initial BigDAWG implementation, we have constructed a number of prototype pieces of the overall architecture of Figure 1. In particular, we have developed basic SHIMs to move data between islands and database engines as well as a simple method that leverages CSV export/import to cast data between engines. In order to further develop BigDAWG, we will make use of detailed query monitoring statistics such as query execution time, machine level statistics and cast execution time. Further, we assume that the datasets we access have defined schemas that exist for all query-able objects.

3.1 Modeling Workloads and Systems

Low latency queries are key for IA and visualizations. From a database perspective, maintaining low query latency can be achieved through different mechanisms such as approximate queries, scaling of server hardware, intelligent result caching and/or optimized database selection. While approximation, elasticity, and proactive caching are active areas of research for BigDAWG collaborators, our current effort concentrates on proper data placement to promote low latency queries.

The same data may be accessed in different ways, for example, analyzed vertically or horizontally. In the long term, the types of analysis will determine where the data will be stored. One example pertains to notes about patients from our motivating medical use case [13]. A typical data access approach may be to display holistic information about a patient, including his or her notes which were written by a physician (vertical analysis). In this case the patient structured data (e.g. age, visits) should be stored in a relational database like PostgreSQL. On the other hand, when a physician wants to find notes that mention a given illness, then a fast text processing should be carried out, and the data should be moved to a text processing engine, such as Accumulo (for horizontal processing). Similarly, the waveform data from medical monitoring devices could be analyzed for a single patient in a relational database or a batch of them should be processed efficiently (e.g. numerical analysis) in an array database, such as SciDB.

To properly place data objects, the system must understand both the types of queries being executed (e.g. the workload, underlying data model) and how a database system is expected to perform the query. We expect the number of supported queries and number of databases to be large. Given this, it will be difficult to come up with pre-defined rules or exhaustive solutions that cover all possible cases. Instead, we propose to use black-box approaches for workload modeling and system performance.

For workload modeling, we must derive strategies to classify and extract features for the wide variety of encountered queries. Examples of broad query classifications include filter and group-by aggregation queries (e.g. histogram), matrix based operations (e.g. linear regression), counting queries, selection queries (e.g. get by ID), or complex joins. Our initial approach to classifying queries will be to use textual similarities in query strings, as well as using exposed database query plans to understand the derived query plan tree. The end goal of this query classifier will be to extract features from a textual query, such as the number of joins, filters, predicate selectivity, and so on. These approaches will rely on supervised learning techniques to bootstrap the system.

With query classes and features, and the ability to CAST data between the systems, BigDAWG will rely on black-box performance profiling of underlying databases. We plan on executing such profiling in three modes. First, in *training* mode, BigDAWG will “replay” every executed query on all systems when system resources are abundant. This will allow the system to build an initial profile of how the systems perform for a given query’s feature and class. Second, in the *opportunistic* mode, BigDAWG records queries to periodically replay on all engines over time in periods of low activity. This allows the system to profile without over-consuming resources, and repeated profiling will help model how consistent query performance is with changing workloads, loads, and hardware. Finally, in the *optimized* mode, BigDAWG will execute each query on one or two systems when resources are constrained. If enough profile data has not been gathered, BigDAWG will randomly select one or two engines to execute the query on. Over time, this will build up a performance model that can be used to select the appropriate engine for using techniques similar to collaborative filtering [5].

BigDAWG can utilize these workload and system performance models to make decisions about query planning when data is replicated, and about long term data placement to ensure that data resides in the ideal system.

3.2 Data Transfers Between Databases

Data movement between engines is a key requirement for BigDAWG applications. For each data object, there may be multiple database candidates within an island, and users are given the abilty to explicitly move data between islands via the CAST operator. Therefore, fast data transfer between databases is a crucial use case for BigDAWG. Designing an efficient solution is difficult due to bottlenecks emerging from disk Input/Output, network Input/Output, or CPU cycles.

To begin exploring the design space of cross-system data migration, we use patient device data as described in Section 3.1 and limit our focus to data transfer between PostgreSQL and SciDB.

Data transfer between databases can be executed on many levels ranging from external connectors to built-in solutions which facilitate direct data transfer from one database to another. The external solutions are flexible but can suffer from low performance. CSV is the most common data format for external files from which data are loaded to databases. In our experience, data loading between these systems is a very CPU intensive process due to parsing and deserialization. This suggests a major accelerator is applying parallelism when loading data. However, while SciDB offers parallel data loading, PostgreSQL does not. The easiest way to harness parallelism in case of PostgreSQL is to partition the data to be transfered, for example, on a primary key or another attribute(s) which would enable simple division. For example, we can generate the output data in parallel using many clients in PostgreSQL and then load the partitions in parallel from many clients connected to SciDB (each client from PostgreSQL can communicate with exactly one client from SciDB). However, there is a drawback. When we load data the first time to PostgreSQL we can enclose creation of a table and data loading in a single transaction. This allows us to eschew excessive logging (which adds about 8% to the loading time). When, we load data in parallel from many clients, we lose the ability to avoid logging overhead.

Built-in solutions that enable faster data transfers require non-trivial changes to the internal design and implementation of data loading and export features. Each database exposes its inherent and specific binary format which either has to be changed inside the database or the binary data has to be transformed from one format to another on the fly during each data transfer. The following experiment motivates the need for optimized solutions.

Figure 2 shows data migration from PostgreSQL to SciDB. The data contains sampled physiological waveforms from the MIMIC II dataset [13], such as electrocardiogram, blood pressure and respirations. The dataset size ranges from 1 to 30 GB in CSV format and contain waveform values of doubles with two dimension columns: patient identifier and millisecond timestamp. We present three approaches to data migration. The first approach exports, transfers, and imports data in CSV format. The second one directly exports binary data from PostgreSQL, then transforms the data to SciDB binary format on the fly and loads the data to SciDB. For the third approach we modify PostgreSQL to immediately generate data in SciDB's binary format such that data is directly loaded to SciDB. Due to reduced parsing and deserialization overhead, the binary migrations become significantly faster than a CSV-based migration.

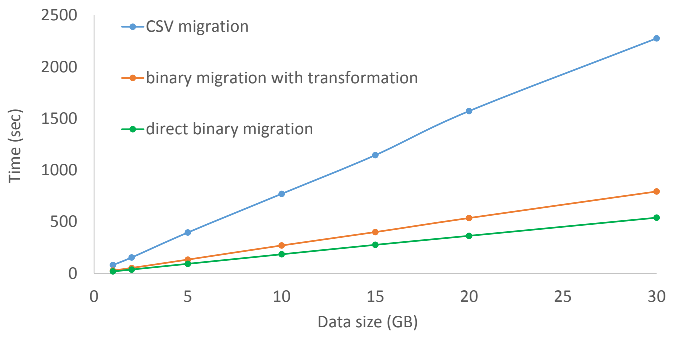

Fig. 2. Data migration from PostgreSQL to SciDB.

3.3 Cross-System Dynamic Replication

As BigDAWG is not limited to one copy of each data object, we plan to explore how to manage replication across systems to improve query performance. Building a dynamic replication technique requires several advances. We plan to explore building models replicating data between systems, using both middleware controlled replication from the API layer, as well as using the CAST operators to synchronize data objects directly between systems. While a streaming CAST operation will likely have lower performance impact on data loading, it will have a reduced level of control as compared with a middleware based approach. With a middleware controlled replication we can on-the-fly track the state of object replication and dynamically scale back how data is replicated in response to either (i) high update rates or (ii) sustained changes in the workload that favor a single engine. In addition to exploring the trade-offs of replication mechanisms, BigDAWG will need to make decisions about which data objects should be replicated, on which engines, and how many replicas should exist for each object.

3.4 Data Transformation from DBMS to JSON

Data in different formats from various databases will need to be unified in one common format for client consumption. We use JSON as a common format which is more concise than XML. A JSON format, combined with the HTTP API exposed by BigDAWG makes it easy for a variety of applications to query data. Other benefits include easier access to data via tools such as cURL, web browsers, new programming languages, and mobile applications. A JSON middleware also solves certain security challenges. For example, it shields the databases from direct and intrusive access.

However, in our initial experiments we found that up to 10% of the total query processing time in BigDAWG is spent on preparation of the final result in JSON format. While this percentage is larger when the underlying data is stored, or cached, in RAM, it can be a significant overhead when low latency is a primary goal. Therefore, a part of BigDAWG will be exploring efficient data transformation to JSON, either through using user defined functions in the database, caching large data objects as JSON in the middleware, or through use of native JSON in the database when available – such as in PostgreSQL.

4 Conclusion

Interactive and web-based analytic applications use data in a wide variety of ways. Our polystore system, BigDAWG, will ease the burden of selecting the right specialized database to ensure low latency queries. In this article, we discuss BigDAWG's architecture and outline several key challenges including fast data migration, workload modeling, dynamic replication, and data transformation.

Acknowledgments

The authors wish to thank our BigDAWG collaborators Kristin Tufte, Sam Madden, Jeremy Kepner, Timothy Mattson, Jeff Parkhurst, Stavros Papadopoulos, Nesime Tatbul, Magdalena Balazinska, Bill Howe, Jeffrey Heer, David Maier, Tim Kraska, Ugur Cetintemel, and Stan Zdonik. This work was supported in part by Intel.

References

[1] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[2] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, and D. Petkovic. Towards heterogeneous multimedia information systems: The Garlic approach. In *Data Engineering: Distributed Object Management*, pages 124–131. IEEE, 1995.

[3] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, et al. S-store: A streaming newsql system for big velocity applications. *Proceedings of the VLDB Endowment*, 7(13):1633–1636, 2014.

[4] S. Chawathe, H. G. Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, 1994.

[5] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, 2013.

[6] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. SIGMOD, pages 1255–1266, 2013.

[7] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(3), 2015.

[8] A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[9] A. Fuchs. Accumulo–extensions to google's bigtable design. *Baltimore ACM Lecture hosted by Morgan State University*, 2012.

[10] V. Gadepally, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Rosa, C. Yee, and A. Reuther. D4m: Bringing associative arrays to database engines. In *IEEE High Performance Extreme Computing*, 2015.

[11] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 881–884. ACM, 2014.

[12] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: souping up big data query processing with a multistore system. In *SIGMOD*, pages 1591–1602, 2014.

[13] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark. Multiparameter Intelligent Monitoring in Intensive Care II (MIMIC-II): A public-access intensive care unit database. *Critical Care Medicine*, 39:952–960, 2011.

[14] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[15] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose time has come and gone. In *ICDE*, pages 2–11, 2005.

[16] M. Stonebraker and L. A. Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[17] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, pages 38–49, 1992.